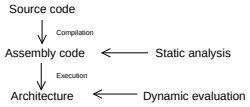
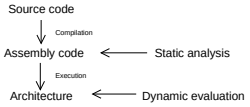
Assessing Application Code Quality

Compilation/Evaluation steps



Assessing Application Code Quality

Compilation/Evaluation steps



Low-level performance analysis

- Assesses the dynamic behavior of compiled code
- But low quality code can have good behavior (missing fma) and difficult to correlate with sources

Static Analysis

- Assess assembly code quality w.r.t. source code
- Detection of inefficient code patterns (missing fma)

Assessing Application Code Quality

MAQAO Goal

- Tool for static analysis of assembly code
- Navigate through assembly and source codes
- Enable user to script more static analyses
- Combine static and dynamic evaluations

Related Tools

- VTune, Caliper, CProf: hardware monitors
- ATOM, Pin: code instrumentation
- Salto: toolkit for low-level code transformations
- HPCview, Finess: address static and dynamic sides

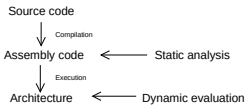
Modular Assembler Quality Analyzer and Optimizer

L. Djoudi, D. Barthou, P. Carribault, C. Lemuet,
J-T. Acquaviva, W. Jalby

PRISM, University of Versailles
LRC ITACA CEA/DAM, University of Versailles
Bull Corporation

Assessing Application Code Quality

Compilation/Evaluation steps



Low-level performance analysis

- Assesses the dynamic behavior of compiled code
- But low quality code can have good behavior (missing fma) and difficult to correlate with sources

Assessing Application Code Quality

MAQAO Goal

- Tool for static analysis of assembly code
- Navigate through assembly and source codes
- Enable user to script more static analyses
- Combine static and dynamic evaluations

Parsing Assembly Files

- Parser generated by scripts from Assembly Instruction Set Manual (240p), fast development
- Ability to regenerate the code
- Includes interesting information given by `icc` through comments:
  - Profiling information
  - Source line numbers
  - Basic blocks
- All information put into a database

Scripting More Analyses

Tap information gathered through SQL queries.

Knowledge Base

- User-defined criteria for code quality
- Application specific criteria

Categories of Analyses

- Gathering statistics (counting nops, fmas,...)
- Histograms (IPCs, block sizes)
- Simple code pattern detection (setf/getf patterns)
- Compiler optimization detection (pipelined loops)

Profiling Assembly Code

Direct Assembly Instrumentation

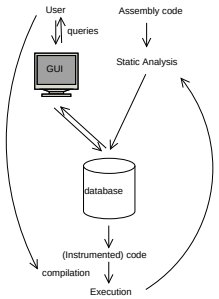
No interference with compiler optimizations

- User selects code fragments to profile
- Keep track of individual execution times
- Monitor:
  - Function parameters
  - Addresses used in load/stores
  - Loop trip count

Overall Structure of MAQAO

Modules

- Assembly parser
- Computing hierarchical structure for assembly code
- Scripted static analyses for gathering statistics,...
- Static/Dynamic performance evaluation
- Dynamic analysis
- Interface for optimizations



Structuring Assembly

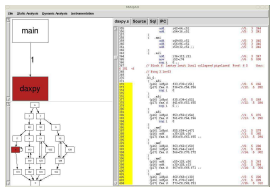
Ease the search and navigation in assembly.

Hierarchy of structures

- Call graph
- Control flow graph
- Blocks, bundles

Navigating through structures

- From source code to assembly
- From call graph, cfg to code



Structure information put into the database

Static/Dynamic Performance Evaluation

Adaptation of MACS Bounds Model [Davidson] for Itanium2

Essential instructions

- Evaluate performance w.r.t. some type of instructions
- Essentials: floating point, memory, branch instructions

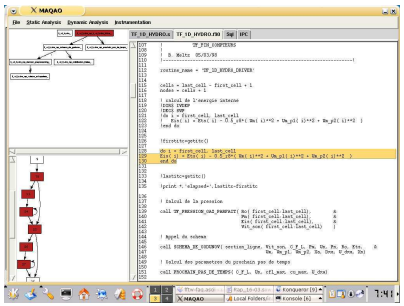
MACS Bounds

Three levels of performance bound:

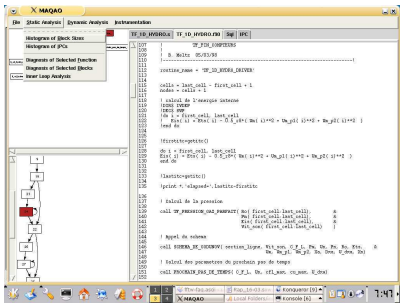
- MAC: counting essential instructions on assembly, no schedule constraint
- MACS: counting instructions with schedule
- Measured performance

Gaps between bounds give hints on what may go wrong.

Some Snapshot: link between CFG and Source code



Some Snapshot: launching some pre-defined requests



Towards Optimization

Not an automatic process. Help user:

- To choose compiler optimization flags
- To rewrite high level code

Export function to optimize to XLG.

Profiling Assembly Code

Direct Assembly Instrumentation

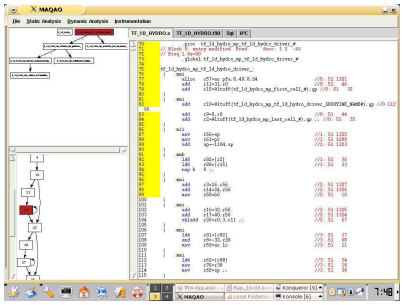
No interference with compiler optimizations

- User selects code fragments to profile
- Keep track of individual execution times
- Monitor:
  - Function parameters
  - Addresses used in load/stores
  - Loop trip count

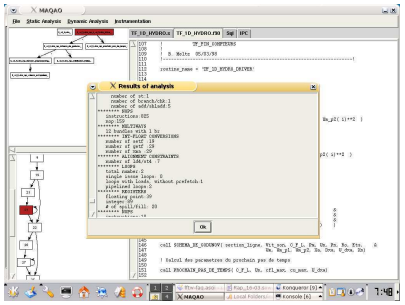
Other Profiling

- Use `prof_gen`, `prof_use` to annotate code
- Resort to `perfmon`

Some Snapshot: link between CFG and ASM code



Some Snapshot: results from some pre-defined requests



Case Studies: Tera Benchmark

Compiled with `icc 8.1, -O3 -fno-alias`.

EIS loop

Original version:

- One loop pipelined with unroll factor 2, epilog pipeline,
- High latency floating point instructions (`setf, getf, xma`) for address computation,
- Useless explicit `spill,fill`.

Case Studies: Tera Benchmark

Totalisation function

Original version:

- Deep pipeline, inefficient for small vectors,
- High latency floating point instructions (`setf, getf, xma`) for address computation.

Case Studies: SHA-0 Attack

Compiled with `icc 8.1, -O2 -fno-alias`. Original version:

- Useless spill/fill separated by more than 600 instructions
- Alignment problems with `ld4/st4`, generating L1 misses

Case studies

Run on BULL Novascale with 256 Itanium2.

- Tera benchmark
- SHA-0 attack

Case Studies: Tera Benchmark

Compiled with `icc 8.1, -O3 -fno-alias`.

EIS loop

Original version:

- One loop pipelined with unroll factor 2, epilog pipeline,
- High latency floating point instructions (`setf, getf, xma`) for address computation,
- Useless explicit `spill,fill`.

Improved version:

- Two versions, unrolling factor of 8, tail code with an explicit switch
- Rewriting function in C, better address computation
- Removal of `spill,fill`.

Performance improvement for this loop: 22%

Case Studies: Tera Benchmark

Totalisation function

Original version:

- Deep pipeline, inefficient for small vectors,
- High latency floating point instructions (`setf, getf, xma`) for address computation.

Improved version:

- Peeling vector loops in C

Performance improvement for this function: 20%

On-going and Future Works

Finding more structure:

- Structuring assembly from C++ codes
- Reaching definition analysis

Optimization side:

- Detection of code templates

Case Studies: SHA-0 Attack

Compiled with `icc` 8.1, `-O2 -fno-alias`. Original version:

- Useless spill/fill separated by more than 600 instructions
- Alignment problems with `ld4/st4`, generating L1 misses

Improved version:

- Replace spill/fill with nops
- Padding for 64bit alignment

Performance speed-up: factor of 2.